Reliable Software Design Aided by QEMU Simulation

Rui Almeida
Department of Industrial Electronics
Centro Algoritmi, University of Minho
Guimarães, Portugal
b12105@algoritmi.uminho.pt

Luís Novais
Department of Industrial Electronics
Centro Algoritmi, University of Minho
Guimarães, Portugal
lnovais@dei.uminho.pt

Nelson Naia
Department of Industrial Electronics
Centro Algoritmi, University of Minho
Guimarães, Portugal
b7191@algoritmi.uminho.pt

Rui Faria
Bosch Technology and Development Center
Bosch Portugal
Braga, Portugal
rui.faria@pt.bosch.com

Jorge Cabral
Department of Industrial Electronics
Centro Algoritmi, University of Minho
Guimarães, Portugal
jcabral@dei.uminho.pt

2021 22nd IEEE International Conference on Industrial Technology (ICIT) | 978-1-7281-5730-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICIT46573.2021.9453486

***Abstract*— Highly reliable systems achieve a low failure probability during their operational lifetime with the help of redundancy. This technique ensures functionality by replicating components or modules, on both software and hardware. The addition of redundancy and further architectural decisions that arise from its usage results in increased system complexity. The resultant complexity hinders analytical approaches to evaluate competing architectural designs, as the time and effort spent with this type of evaluation may significantly delay development. A way to avoid time spent on this type of analysis is to submit the designed architecture to simulation, both for validation and evaluation. In this paper, we propose the usage of a simulation tool, specifically QEMU, to assist reliable system development and simulation. Based on this tool, extensions were developed, aiming for a simulation environment that covers the redundancy use case, allowing to validate the complex interactions under redundant architectures, and supports reliability estimations to compare architecturally redundant designs.**

Keywords—reliability design and estimation, co-simulation, QEMU, redundancy

I. INTRODUCTION

Embedded systems cover applications ranging from General Purpose systems, such as household electronics, to Safety Critical systems such as flight and nuclear control [1]. The development of Safety Critical applications requires particular attention since system failures can incur on significant economic loss or possibly human lives. The possibility of disasters under such type of applications brought system reliability concepts to the foreground. Reliability is a system metric that is directly related to its life expectancy, meaning that highly reliable systems present the highest times before system failure. High reliability systems are mostly found in the fields of avionics [2][3], life support [4], and more recently in the automotive sector [5].

One way to increase system reliability is to replicate system components, allowing it to achieve an higher time before failure, consequently reducing its failure probability. This replication technique is known as redundancy, and it can be implemented on both software and hardware. The usage of redundant architectures is connected with an increase in cost and complexity as well as synchronization problems [6]. This is the main reason why both hardware and software architectures must manage redundancy well. A redundant architecture can have several processing modules, communicating with each other and making decisions about the system operating state. This implies that redundant modules present at least one channel of communication between them. Lack of solid synchronization mechanisms can disrupt the interactions between redundant systems, defeating the purpose of redundancy.

The addition of redundancy is not the major issue on reliable designs, as most complexity comes from the architectural decisions made regarding the redundancy under application context. Such decisions not only regard the hardware to replicate, but also the software that manages it, greatly increasing complexity. Under this perspective, the solution space is vast, and can be difficult to evaluate which architectural decisions are better for the context. Even after deciding about the level of component redundancy to apply, there is a wide variety of architecture decisions on how the redundancy is used, and on how it will affect system behaviour. The complexity can be so high, that an analytical approach to evaluate the design may be time consuming and ineffective. Furthermore, the increase in complexity can make system weaknesses less apparent, since particular behaviours can be harder to identify, making it difficult to exactly pinpoint the problem under such complex systems. With this in mind, it may be beneficial to implement the design and submit it to simulation-based evaluation, in order to decide on what solution to adopt, under the vast solution space.

This article aims to take a pragmatic step into a simulation environment that aids reliability development by providing mechanisms to not only validate but also evaluate the global system reliability of architectural decisions made in the context of redundancy. With that in mind, QEMU was supplemented with three extensions that allow for both multi-modular processing system simulation, tackling problems that arise from

978-1-7281-5730-6/21/$31.00 ©2021 IEEE

this type of simulation, and reliability estimations through simulation-based techniques.

II. Background and Related Work

This section addresses topics that help to understand the methodologies used and the theoretical background behind the simulation environment proposed: (1) the concept of redundancy; (2) QEMU as a full system emulator and as the simulation tool chosen for the extensions; (3) co-simulation as a technique to simulate redundant modules; (4) reliability metrics and what they mean; (5) estimation of reliability metrics and techniques that support them.

A. Redundancy

Redundancy is a technique that rests on having extra components designed to have the same functionality as the original ones. By adding these redundant components, or replicas, it is ensured that if some part of the system fails, a redundant component resumes the functionality of the faulty one. There are two kinds of redundancy: spatial and computational. Spatial redundancy provides additional components, functions, or data items to mask faults that may happen on the original components. Space redundancy is further classified into hardware, software, and information redundancy, depending on the type of redundant resources added to the system. In computational redundancy the computation or data transmission is repeated, and the result is compared to a stored copy of the previous result.

One of the common forms of redundancy is hardware redundancy. Hardware redundancy is when two or more physical copies of the hardware component, or system module, are used, performing some of the functions already provided by the original system. A natural evolution of hardware redundancy consists of having two or more component replicas operating in parallel. Dual Modular Redundancy (DMR) is a common solution, where two processing units execute the same task at the same time and communicate between them to compare results and detect any possible faulty behaviour. Following this architecture, n-modular redundancy (n-MR) tries to mitigate the intrinsic problem of error correction of DMR architectures, by adding more modules and a voting entity. The main issue with this technique is that similar architectures can respond equally to faults, which may cause the voter to produce erroneous results.

Redundancy can also be classified as homogeneous or heterogenous, depending on the type of redundant modules used. In homogenous redundancy, the same technology is replicated to perform the same function, mitigating only random failures [7]. On the other hand, the heterogeneous approach uses different technologies to perform the same function, allowing the system to recover from systematic failures due to a given technology's inherent limitations

B. QEMU

Quick EMUlator (QEMU) [8] is an open source full system emulator that translates target binary code to host binary code. It features the fast emulation of several CPU architectures (e.g., ARM, x86, Sparc, Alpha) on several host platforms (e.g., ARM, x86, PowerPC). Being an open-source software, its source code can be changed in order to edit its features or even add new ones, to achieve developer's needs.

The hardware emulation is done with the help of models that mimic real hardware behaviour. These models respond to write and read operations during code execution, by using functions and routines that contain an approximation of how the hardware would respond to those operations. Although the behaviour is emulated, latencies specific to such operations cannot be emulated.

One of the features that makes QEMU different from other full system emulators is that binary translation does not occur on instruction level, but instead guest code is split into translation blocks. These translation blocks contain several target instructions and are executed atomically. This accelerates a typical slowdown from constant overhead of executing instructions one at a time. The execution of the translation blocks dictates the time advancement during the simulation, which is proportional to the number of instructions executed.

QEMU has already been used for wide variety of research purposes. Under the reliability topic, it has been used, by several authors, for fault injection and software metric estimations [9][10]. At the time, there is no evidence of QEMU being used to approach the redundancy use-case under reliability development.

C. Co-simulation

Typically, within a complex system, models developed in different domains are independently validated, meaning that no real interactions exist between them. Although testing is independent, the models need information from other domains to have meaningful simulation results. On this context, co-simulation is a technique to simulate several domains and the interactions between them. It consists of enabling global simulation of a complex system through composition and interfacing of simulators from different domains.

Although this may seem an excellent way to validate a complex system, the act of simulating in different domains makes interactions difficult due to the different temporal abstraction levels. Since different domains simulators run on different abstraction levels, the time granularity may be different across simulators, which means, at a given wall-clock time, the simulators may all have different simulation times, with different execution advancements. For this reason, simulations need to obey a synchronization mechanism since simulations are independent and interactions should be correctly timed for both simulations.

Under the synchronization context, there are two major classes of synchronization [11]: (1) conservative, which strictly avoid causality errors, and (2) optimistic, which allow causality errors and recovers from them. Two well know algorithms are the Chandy-Misra [12] for conservative methods, and the Time Warp algorithm [13] for optimistic.

Conservative synchronization is based on the work of Chandy and Misra in which events are processed in sequential chronological order and simulations exchange time-stamped messages. These mechanisms assure that all messages are attended on time. To do so, the simulations are blocked from further processing until the next message can be safely sent and

received on both simulations. The main issue of any conservative simulation is determining how much can a simulation can execute to avoid any causality error.

On the other hand, optimistic synchronization algorithms allow causality errors to happen and recover from them. If a causality error is detected, the simulation must be rolled back, meaning that all preceding simulation results must be undone until the causality error is resolved. Before the occurrence of a causality error, the simulations are not synchronized and run independently of each other, therefore only being synchronized when causality error occurs.

D. Reliability Metrics

System reliability is a quantifiable metric which estimates the expected useful life of a system. It is given by a cumulative distribution function:

$$R(t) = 1 - Q(t) = 1 - \int_0^t f(\tau)d\tau = \int_t^\infty f(\tau)d\tau \qquad (1)$$

where Q(t) is the unreliability function, which defines the probability of failure over time. Subtracting this probability from 1, gives the reliability function.

Another expression that is always part of reliability is the Mean Time Between Failures (MTBF) which is expressed as:

$$MTBF = \int_0^\infty \tau \cdot f(\tau)d\tau \qquad (2)$$

where, τ is the time in hours and f(τ) is the probability density function of failure.

E. Reliability Estimation

Although reliability metrics can be calculated by traditional methods, complex systems with large number of different components makes calculation impractical. Some techniques and methods can be used to synthetize a prediction of such metrics, avoiding the inevitable hard and time-consuming work that normally would come with traditional methods. The Monte Carlo method fits the available techniques by providing numerical estimation of an unknown parameter or metric by the mean of repeated sampling.

An example of a Monte Carlo simulation in reliability engineering context to evaluate system failure probability, consists in running many repetitive trials and changing component states according to a probability distribution. On each one of these trials, there are time steps which represent advancement of system lifetime. On each of these steps, component states are changed by generating random numbers and comparing them to the component's failure distribution. If the random value is lower than the component failure rate at the given time, component state is changed, otherwise no change is made. The process is done until system failure, and the resulting time step is stored. As the trials are done several times, the time step results of each trial will create a system failure distribution, which approximates the real system failure rate, with an uncertainty.

Under reliability engineering, this method can also be combined with fault injection techniques, in order to gather insight about system-level behaviour. The goal of this technique is to provoke (inject) faults, or stimuli, that are as close as possible to real faults that could occur on real hardware. By injecting faults into a running system, it can provide information about the failure process, which means that metrics such as mean time between failures (MTBF) can be taken from fault injection results.

III. QEMU Extensions for Reliability

This section describes the simulation extensions that were developed with the main goals of multi-modular processing system software validation and reliability estimation with fault injection. With all this in mind, three extensions for QEMU were developed. The Synchronization extension aiming to mitigate causality errors during simulation of redundant modules. The Shared Bus extension that allows for redundant modules to communicate with each other. The Fault Injection extension enabling reliability estimation capabilities, by providing mechanisms to inject faulty stimuli to system components.

A. Conceptualization

Under a redundant architecture, a target system can have multiple redundant subsystems, each contributing for the output. The redundant modules have independent hardware and computations are made within each subsystem processor. Each redundant module is conceptualized as a QEMU instance (or simulation), running all the software stack and emulating all the hardware that composes the subsystem. The diagram in Figure *1* presents the conceptualization of the redundant modules as QEMU instances.

The redundant subsystems use the same inputs to compute a value which contributes to the system output. The computed values should be equal across subsystems if all of them do the same operations and receive the same inputs. This happens when the system presents an homogeneous architecture. Such behaviour differs from heterogenous architectures which can have different inputs and outputs between subsystems. Alongside data output and computation, redundant subsystems may also be connected between them for N connections depending on the number of redundant subsystems present.

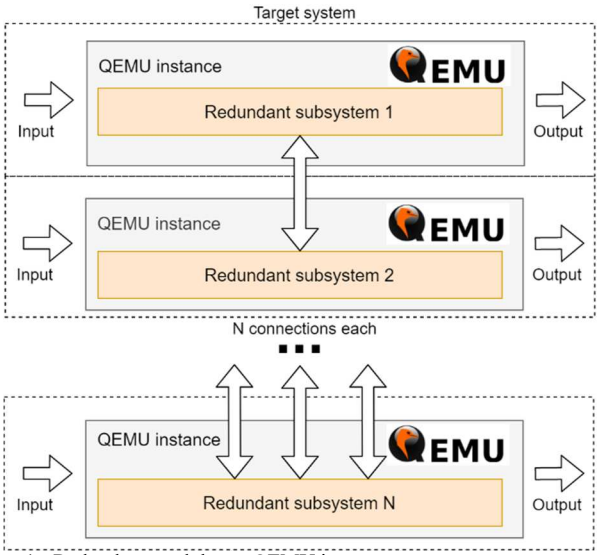

Figure 1 - Redundant modules as QEMU instances

As the instances are independent from each other, there must be guarantee that causality errors are attended to. Consequently, a synchronization mechanism was implemented to mitigate any simulation synchronization issues between redundant subsystem simulations.

B. Synchronization

For simulation synchronization, a conservative method was implemented, based on a time budget concept, which guarantees that no causality errors happen. The diagram on Figure *2* describes the time budget concept. The simulations run for a specific amount of time (or budget) and are then blocked from executing further, until all other simulations reach the same simulation time. For this purpose, both a process was created, which handles synchronization between simulations, and QEMU was modified to obey to this same process and execute target code in a time budget manner.

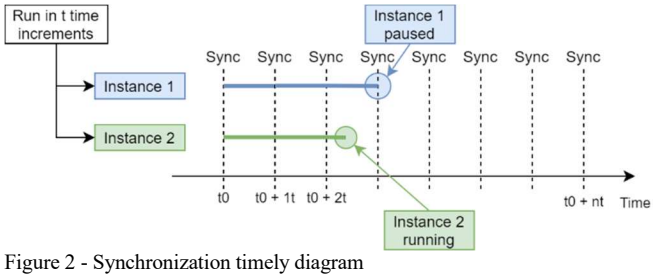

Figure 2 - Synchronization timely diagram

The process listens to incoming simulations to be synchronized and handles the synchronization requests. When an "out of budget" message is received, meaning that a simulation reached the end of its executing budget, the process checks if all other connected simulations have already reached the same simulation time. On that case, all blocked simulations resume execution. If any simulation is still running, all simulations are blocked from further execution until all reach the same simulation time.

On QEMU, the target code execution algorithm was modified to force simulations to run for a time budget that is application specific and user specified. During emulation, the execution of translation blocks and further time advancements are monitored to assure increments within the time budget. Every time advancement contributes to the depletion of the available time budget and upon exhausting it, simulations block execution and send an "out of budget" message to the synchronization process. The process then replies with a "Resume" message, which allows the simulations to resume execution up until the duration of the next time budget.

Time budget granularity depends on the time per instruction, which is directly related to the *icount* parameter chosen for the emulation. The lowest number possible for the time budget is the time to execute one instruction, meaning that, in this case, synchronization would occur at instruction level.

This type of implementation does not limit the extension usage to QEMU simulations only. Any simulation tool that can run in a time budget manner can be introduced into a simulation environment and be correctly synchronized with different tools from different domains.

C. Shared Bus

Under a redundant architecture, communications and consequent interactions are typically associated with communication peripherals such as LPUART or SPI modules. The Shared Bus extension aims to expand emulated communication peripheral's capabilities to allow interactions between different processing subsystems. The extension emulates a data bus and its transactions, covering typical R/W operations done by common protocols such as UART, while at the same time allowing for multiple peripherals to connect to it, attending to the bus characteristics of more complex protocols such as CAN or SPI. This kind of emulation entirely abstracts the communication protocol's timings and working principles. It only concerns the communication behaviour by saving the data and relaying it to other communication peripherals.

As such, the developed extension is composed by two parts: (1) A process (called Shared Bus) that manages all communication connections to an emulated bus; (2) A node interface allowing peripherals (and other tools) to interact with the emulated bus for read and write operations.

The Shared Bus process manages peripheral connections and data transfers. Connected peripherals can perform write and read operations on the emulated bus, which saves the data written by the peripherals. When a write operation occurs, the written data is saved and relayed to all peripherals connected to the bus. It is the connected peripherals responsibility to attend the relayed messages, consider invalid data and check the integrity of the data received.

The peripherals and the Shared Bus process communicate with each other through a pair of sockets in a client-server configuration. Each peripheral uses the sockets as following: (1) A client socket used for synchronous read/write operations; (2) A server socket used for asynchronous reads from the peripheral. Peripheral asynchronous reads can be emulated by attending the relayed data upon write operations by other peripherals, and for that reason peripheral implementation requires a mechanism that allows to always listen to these asynchronous events (such as a thread). This mechanism mainly emulates typically ISR triggered read operations. An example usage of the extension is presented in Figure *3*. It shows an emulated CAN bus connected to emulated CAN peripherals from different QEMU instances.

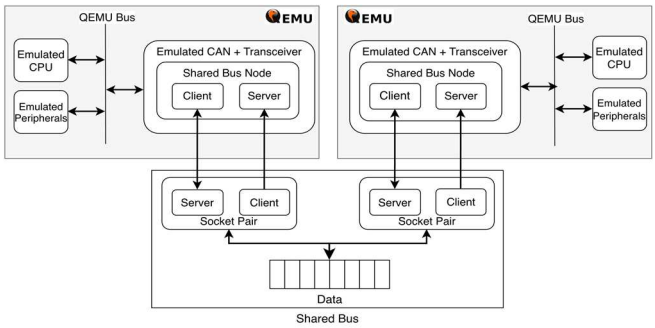

Figure 3 - Shared Bus extension diagram

D. Fault Injection

As mentioned before, fault injection can be used to evaluate a system, by provided quantifiable reliability metrics. Under that perspective, QEMU was extended to allow fault injection capabilities on different system components. The extension is based on the research of Andrea Höller [9], where a framework, namely FIES, aiming to assess software fault tolerance was developed.

The fault injection extension integrates some components from the FIES framework and adds an external coordinator, as shown in Figure 4. Each of the components are described next.

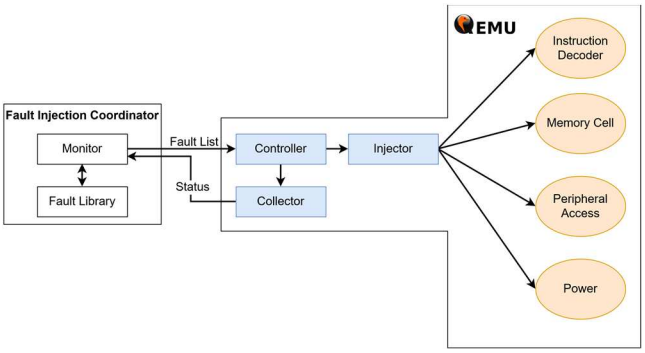

Figure 4 - Fault Injection extension components

Fault Injection Coordinator: this entity generates faults and creates a fault list with them. Within the coordinator, the monitor manages simulation connections and controls simulation experiments based on the data from the Collector. The Fault Library contains the possible user-specified faults to be used on any fault experiment.

Controller: decides how to inject faults according to the fault list. The faults specified on the fault list remain on the system for a user-specified amount of execution time. Based on this information and the QEMU built-in timer, the controller decides when and where a fault should be triggered or stopped. The controller also parses the fault list, which comes as an XML file from the coordinator.

Injector: core of the fault injection. It contains functions and methods that allow injection of the different types of faults. Faults can be of four different types which occur on different execution locations: instruction decoder, memory cells, peripheral access and system power. According to the faults present on the fault list received by the controller, the fault specific functions are called.

Collector: gathers information about the status of the simulation after any fault is injected. The goal of this component is to gain knowledge on how the system responds to the fault by retrieving the system's internal execution status using monitor variables.

As previously mentioned on the fault injector description, faults can occur on instruction decoding, memory cells, peripheral access or system power. Besides these type of faults, a clock fault type was also added, which is a type of fault addded to cover a particular behaviour of the case study. Each one of these type of faults will be addressed next.

Instruction Decoder: these type of faults occur before execution of translation blocks, when the target code is disassembled. This fault replaces the current disassembled host instruction, overwritting it by the new instruction defined in the fault list.

Memory Cell: occurs during read and write operations on physically addressable memory. Such operations are monitored and as they are realized, available memory faults are injected. When a read or write operation is done, the value written or read is overwritten by the value specified in the fault list.

Peripheral Access: prevents access to peripheral memory regions by the QEMU system bus, rendering a specific peripheral unusable. This is done by monitoring memory access during both read and writes on the QEMU system bus and blocking any access that matches the fault specified address.

Power: aims to simulate a power failure on the system, forcing the QEMU instance to reset the CPU and every peripheral. Although this resets the simulation, the total simulation time is not affected, as QEMU keeps track of the simulation time up until shutdown of the instance. This type of fault is particularly important for redundant systems, since it allows to evaluate the system behaviour when a redundant module shuts down.

Clock: this is a special type of fault added for the case-study. This was added aiming to emulate clock drift type of situations between redundant subsystems. Since QEMU does not emulate real clock timings, real clock speed drifts are not possible to represent. With that in mind, and knowing that all simulations obey to the synchronization process, a clock fault means loss of synchronization between a simulation and the synchronization process. This is done by dropping the communication between them and letting the simulation run at its own pace.

Externally to QEMU, as previously mentioned, the Fault Injection Coordinator handles simulation experiments. The implementation of the coordinator is not generic and it is the developer burden to implement it in a way it satisfies the simulation needs. This is because the possible faults and simulation management decisions are case-study specific and depend on what the user wants to observe as simulation result.

IV. Case Study

The developed extensions were used on the validation and reliability evaluation of a case-study that presented mission-critical characteristics. The system is responsible to acquire sensor data and output it to a system bus. It presents hardware homogeneous redundancy, as it is composed by two identical subsystems that output the same type of data (Figure 5).

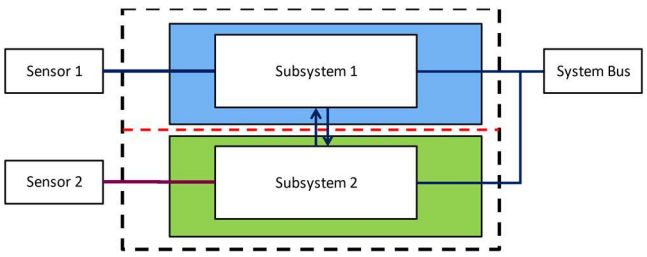

Figure 5 - Case-study concept

The main system requirement is to provide processed sensor data at a rate of 10 milliseconds. Each subsystem outputs processed sensor data every 20 milliseconds with 10 milliseconds offset between them, assuring a 10-millisecond data throughput from the system. Upon subsystem failure, the remaining functional subsystem reconfigures itself to guarantee the initial throughput requirement. Feedback between subsystem is made through a communication channel, allowing to correctly synchronize subsystem actions.

The software is responsible to manage the subsystem redundancy by deciding the subsystem operating state depending on the feedback received. During runtime, both subsystems have specific time windows to provide feedback about their current operating state. The lack of feedback within that same window is assumed to be a subsystem failure, forcing the operational subsystem to reconfigure itself.

The platform used in each subsystem was S32K116 from the S32K family from NXP. The platform is based on a 32-bit ARM Cortex M0+ machine within a SoC that is specially designed for Automotive applications. Due to the lack of platform native support on QEMU, the platform was created, along with its peripherals and *on-board* devices, and added to the QEMU supported machines. Furthermore, the peripherals that allow interactions between subsystems were extended to interact with the Shared Bus extension.

One of the drawbacks of this type of simulation environment under QEMU, lays on the models used for the emulate peripherals. Any design or implementation bug on such models can be misinterpreted as a real hardware limitation, contributing for wrong emulation results.

A. *Simulation Environment*

The simulation environment that allowed to validate the case study is shown in the figure below. The environment combined two QEMU instances, one for each redundant subsystem, two Shared Bus instances, for two different communication types, and one synchronization process.

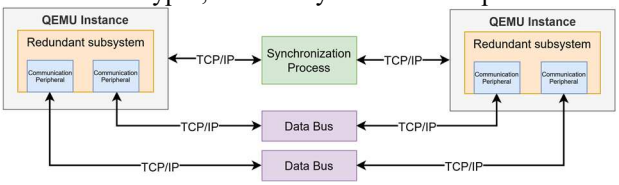

Figure 6 - Simulation environment used for case-study validation

Validation went through analysing the timestamps of messages sent to the Shared Bus and through monitoring of internal program variables. This validation was made under a *Linux* environment, as seen in Figure *7* and Figure *8*. The figures demonstrate the entities used in the simulation environment. On the right side of the figure, the two terminals are the QEMU running instances that emulate each one of the subsystems. The terminals output internal state variables to have an insight of the system running state. On the bottom left side, the synchronization process shows the time increments of both instances. This value matches the user chosen value of 50016 nanoseconds for the time budget. On the top left, the data bus shows messages that are sent from both redundant subsystems, alongside with the timestamp at which they were sent. As seen in the figure, the messages are sent every 10 milliseconds, validating the initial system requirement.

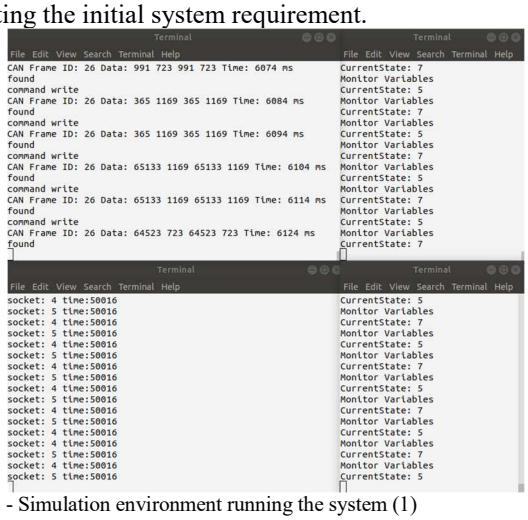

Figure 7 - Simulation environment running the system (1)

Upon confirmation of the main requirement, the redundancy management was validated by mimicking subsystem failure. Upon aborting one of the QEMU instances, the remaining subsystem correctly reconfigured itself to ensure the initial 10ms data throughput, as seen in Figure *8*.

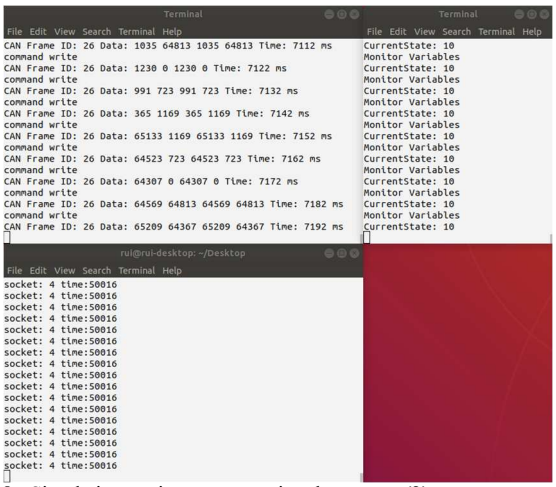

Figure 8 - Simulation environment running the system (2)

B. *Reliability Estimations*

In order to evaluate the system for its reliability metrics, estimation of such metrics was done during simulation, supported by the extensions developed. The system went through Monte Carlo simulations, injecting faults into system component blocks, according to their failure distributions. Since there was not enough data to get an accurate probability of failure distribution of each component, failure rate probability curves were created according to the MTBF values of the components (Table *1*).

Table 1 – System components MTBF values

Component	MTBF (hours)
Microcontroller	4.2×10^5
Communication Module 1	1.7×10^5
Communication Module 2	2.5×10^5
Clock	1.6×10^5
Power	5.3×10^5
Sensor	2×10^5

The curves were parameterized to fit a Weibull distribution, alongside with a cumulative failure rate of $1\ x\ 10^{-4}$ in the first 100 hours. The simulations executed until system failure or up until a total simulation time of 1 500 000 hours of component lifetime.

The simulations followed a similar environment as the validation, alongside the Fault Injection Coordinator controlling simulation trials. The resulting environment is shown in the figure below.

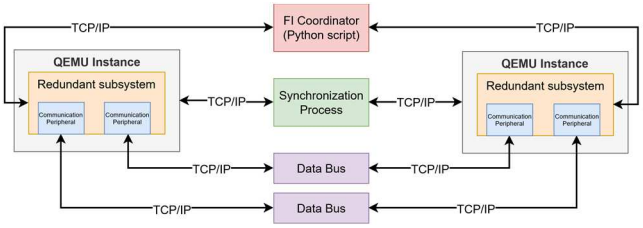

Figure 9 - Simulation environment used for case-study reliability estimation

During simulation, faults were injected until system failure, which was known by checking the status of both simulations from the monitored variables. If both simulations were on failure state, the current simulation trial was over and the final emulation time was saved, as it was considered system failure. When this happened, a new trial started by restarting simulations. If the system was still operating, the trial continued by generating new faults until system failure. New faults were created by generating random values during runtime and checking them against the fault probabilities. Regarding simulation speed, each 10-hour increment of component lifetime (or Monte Carlo step) during the trials took approximately 75ms of wall-time to complete.

C. *Results*

The simulations resulted in 181 trials, 10 of which reached the maximum simulation time, with the remaining ones resulting in system failure. From the collected data, the resulting system mean time to failure value was greater than 378797 hours. The resulting distribution of the times before failure is presented in Figure 10.

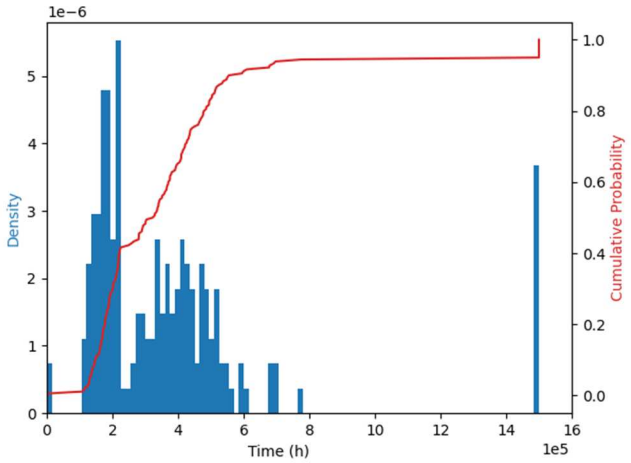

Figure 10 - Probability density and cumulative density functions of the simulation results

Although the system presented such time before failure, sensor faults caused the system to output wrong data while still being in an operational state. In order to get an overview about the time the system outputs wrong data while being operational, the distribution in Figure *11* shows the data relative to such behaviour. The resulting mean time before wrong data output was 194361 hours.

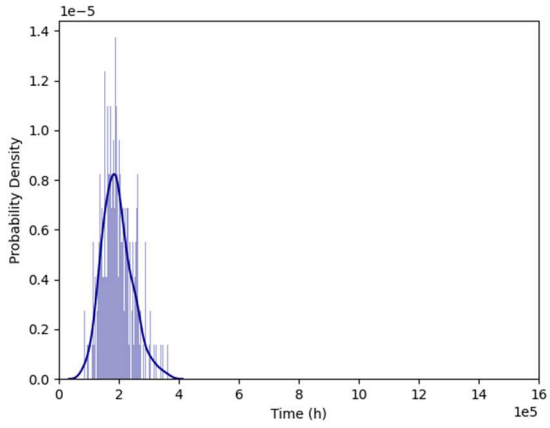

Figure 11 - Distribution of the probability of wrong data output by the system

Since the amount of simulation data was low comparing with the original plan of 10000 trials, ensemble methods were applied to try to gather a better approximation of what the real data would been like. First, the bagging method was applied to the original time before failure data. From the original data, 10000 sets were bootstrapped, resulting on data with a mean value of

379148 hours and a standard deviation of 36302. On the resulting data from bagging, a boosting method was applied, which resulted on a mean value of 338910 hours and a standard deviation of 14566 hours. Under the same context, the data regarding the times before wrong system output went through the bagging and boosting process. First, bagging was applied, generating 10000 sets of 30 samples resulting on data with a mean value of 194392 hours with a standard deviation of 9804 hours. Then, boosting was applied, following the same algorithm and number of iterations used on the time before failure data. The process resulted on a data distribution with a mean value of 184991 hours with a standard deviation of 4149 hours.

Regarding fault occurrence, the histogram in Figure *12* shows the number of faults occurred before system failure. Since there were two redundant subsystems, the maximum possible number of faults was 12, since 6 types of system block faults could happen in each module. Since the system has components that do not contribute to system failure i.e., faults on such components are not destructive for the system, it tolerated a significant number of faults before failing.

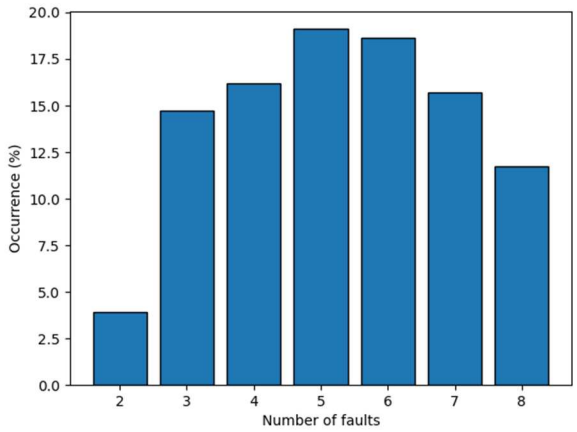

Figure 12 - Fault occurrence before system failure

V. Conclusions

The simulation environment that resulted from the adoption of QEMU complemented with the developed extensions, assists reliability-oriented development, by providing means to validate and evaluate architecturally redundant designs. This way, different complex redundant designs can be evaluated and compared, before adopting a final solution. Beyond avoiding complex analytical analysis to evaluate designs, the design iterations and respective software stacks can be validated before any physical prototype is available, reducing the overall development effort and time. The used simulation tool, QEMU, showed itself very versatile, having a lot of potential to be used as an exploratory tool for simulation-oriented research.

VI. Acknowledgment

This work is supported by: European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 037902; Funding Reference: POCI-01-0247-FEDER-037902].

References

[1] A. Birolini, *Reliability Engineering*, vol. 34, n. 4. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017.

[2] T. Wilfredo, «Software Fault Tolerance: A Tutorial», 2000.

[3] D. Siewiorek e P. Narasimhan, «Fault-tolerant architectures for space and avionics applications», *NASA Ames Res.*, pp. 1–19, 2005.

[4] P. O. Wieland, «Living together in space: The design and operation of the life support systems on the International Space Station», *NASA Tech. Memo.*, vol. 1, n. 206956, 1998.

[5] C. Huang, F. Naghdy, H. Du, e H. Huang, «Fault tolerant steer-by-wire systems: An overview», *Annu. Rev. Control*, vol. 47, pp. 98–111, 2019.

[6] A. Dasgupta e J. M. Hu, «Hardware reliability», *Prod. Reliab. Maint. Support. Handbook, Second Ed.*, pp. 95–140, 2009.

[7] E. Dubrova, *Fault-Tolerant Design*. New York, NY: Springer New York, 2013.

[8] F. Bellard, «QEMU, a fast and portable dynamic translator», *USENIX 2005 Annu. Tech. Conf.*, pp. 41–46, 2005.

[9] A. Höller, «Advances in Software-Based Fault Tolerance for Resilient Embedded Systems», 2016.

[10] Y. Li, P. Xu, e H. Wan, «A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing», pp. 123–127, 2013.

[11] S. Jafer *et al.*, «Synchronization methods in parallel and distributed discrete-event simulation», *IEEE Trans. Comput.*, vol. 77, n. 1, pp. 257–260, 1990.

[12] K. Mani Chandy e J. Misra, «Distributed Simulation: A Case Study in Design and Verification of Distributed Programs», *IEEE Trans. Softw. Eng.*, vol. SE-5, n. 5, pp. 440–452, 1979.

[13] D. R. Jefferson, «Virtual Time», *ACM Trans. Program. Lang. Syst.*, vol. 7, n. 3, pp. 404–425, 1985.